# Stick a *fork* in It:<br>Applications for SystemVerilog Dynamic Processes

Doug Smith and David Long, PhD

Doulos
Austin, TX

Doulos
Ringwood, UK

www.doulos.com

**ABSTRACT**

*In Verilog, processes come in the static form of always and initial blocks, concurrent assignments, and the fork..join statement. SystemVerilog introduces dynamic processes in the form of new fork..join statements and the std::process class. This paper will explore the many applications of dynamic processes in verification and behavioral modeling such as how verification methodologies create independently executing components and control simulation phasing, isolating random number generators for test reproducibility, parallelizing testbench interaction with DPI code, and a novel approach of using dynamic processes with SystemVerilog interfaces to create bus resolution functions and model analog behavior.*

# Table of Contents

## Table of Figures

# 1 Introduction

Hardware engineers have always had to think about concurrency in RTL designs. All operations occur inside of a process and those processes operate concurrently (even continuous assignments are an implicit process). Verification environments are built the same way with the added advantage of using dynamic processes, which allow them to dynamically change and react to the design under test.

There are two kinds of processes in SystemVerilog: *static* and *dynamic.* The SystemVerilog LRM defines a static process as one where "each time the process starts running, there is an end to the process." Another way of putting this is that static processes are created when the code is elaborated and persist until the end of simulation. Static processes come in several forms—each *always*, *always_comb*, *always_latch*, *always_ff* and *initial* procedure is a separate static process as is every concurrent signal assignment.

On the other hand, dynamic processes are created at run-time and execute as independent threads from the processes that spawned them. They can be waited upon or disabled. Dynamic processes come in the form of *fork..join_all*, *fork..join_none*, and dynamic processes created by concurrent assertions and cover properties. Dynamic processes allow a testbench to dynamically react to a design under test, control the flow of simulation, build high-level models, and respond to both testbench components and the design.

Using a *fork* statement is the most common way to create a dynamic process. The traditional Verilog *fork..join* creates run-time threads of execution, but is still considered static since these threads are part of the flow of their parent process and have a clear end point—where all the threads join back together. With the *fork..join_any* statement, the parent process blocks after spawning dynamic processes (one for each forked statement), until one of the *fork* statements complete (Figure 1).

```
initial
begin
    fork
        task1;          // Process 1
        begin           // Process 2
            task2;
            task3;
        end
    join_any
```

**Figure 1 - fork..join_any.**

With *fork..join_none*, the parent process continues execution after the forked statements without waiting for the forks to complete. However, note that the dynamic processes do not get spawned until the parent process reaches a blocking statement (Figure 2).

```
initial
begin
    fork
        task1;          // Process 1
        begin           // Process 2
            task2;
            task3;
        end
    join_none
```

**Figure 2 - fork..join_none.**

To synchronize again with the forked threads, a *wait fork* can be used, which waits not only on the forked dynamic processes but also on any of their children dynamic processes. Similarly, child forked dynamic processes can be prematurely stopped by using *disable fork*. This can be useful to shut down testbench components or end simulation.

Sometimes, we may want to pass a unique identifier to each forked thread to distinguish between processes. Since dynamically forked processes start execution after the parent process blocks or evaluates in the current time step, passing a unique identifier to each thread is not as easy as it might seem. For example, in Figure 3 the *for* loop's index is passed to each forked thread:

```
initial
begin
  for ( int i = 0; i < 3; i++ ) begin
    fork
        begin
            int id = i;
            ...
        end
    join_none
```

**Figure 3 - Loop variables in a fork..join_none.**

The *for* loop spawns 3 threads and passes the value of *i* into each thread, but since the threads do not start until the parent process blocks, each thread begins after the *for* loop has finished its iterations; therefore, each process will have the same *id*—a value of 3! To overcome this problem, *fork* statements may have an optional automatic variable initialization statement and each spawned process will receive a unique copy of the variable as shown in Figure 4.

```
initial
begin
  for ( int i = 0; i < 3; i++ ) begin
    fork
         automatic int id = i;
         begin
            $display("val=%0d", id ); // Each process
                                      // receives unique id
         end
    join_none
end
```

This produces the output:

```
val=0
val=1
val=2
```

**Figure 4 - Correct way to read loop variables in a fork..join_none.**

While not yet supported by all EDA tools, SystemVerilog also defines a very useful process control class defined in the implicit *std* standard package. The *process* class definition is given in Figure 5.

```
class process;
  enum state {
    FINISHED, RUNNING, WAITING, SUSPENDED, KILLED };
  static function process self();
  function state status();
  function void kill();
  task await();
  function void suspend();
  task resume();
endclass
```

**Figure 5 - The SystemVerilog process class.**

This class provides fine-grain control of all processes in the simulation (both static and dynamic), including the ability to suspend, start, wait on, and kill any threads of execution much like job control in an operating system. The *self()* method returns a unique process id that is particularly useful in several applications that will be described in the next section. The example given in Figure 6 demonstrates some of the flexibility and control that the *process* class offers.

```
initial
begin
   process id[10];
   for (int i = 0; i < 10; i++)
      fork
         automatic int j = i;
         begin
            id[j] = process::self();
            /* Do something */
         end
      join_none

   for (int i = 0; i < 10; i++)
      wait ( id[i] != null );

     /* ... */

     for (int i = 0; i < 10; i++)
       if ( id[i].status != process::FINISHED )
          id[i].kill();
end
```

**Figure 6 - Use of the process class to manage process execution.**

Together, static and dynamic processes provide the much-needed tools to construct sophisticated testbenches for even the most challenging verification problems and solve the many modeling problems faced. This paper will present a selection of practical applications for processes and how to benefit from a dynamic multi-process modeling approach.

# 2 Applications

## 2.1. Hardware modeling

While we may not think of using dynamic processes to model hardware, there are two specific cases where using a *fork..join* structure happens to be quite useful. Both applications involve SystemVerilog interfaces, and while the first application does not exactly fit within the definition of a dynamic process, it still appropriately fits within the discussion of using *fork..join*.

### 2.1.1. Extern fork-join in an interface

Our first application of a forked process is in a behavioral model of a bus using tasks exported from a module within a SystemVerilog interface. Interfaces are commonly used in testbenches to provide a primitive Transaction Level Modeling (TLM) communication mechanism between the modules that implement the verification components (also known as Bus Functional Models or BFMs). Modules call tasks or functions in the interfaces to perform transactions, which could otherwise entail pin 'wiggles' over many clock cycles. These tasks and functions are imported into the modules via a *modport* in the interface. One significant disadvantage of this approach is that moving a transaction from an initiator to a target module requires processes in both initiator and target: the initiator must call a task or function to transfer the transaction to the interface; the target must then call a task or function to fetch the transaction from the interface. If a bus protocol requires handshaking or if a response must be sent from the target back to the initiator then a considerable number of tasks/functions may be called for each transaction, complicating and potentially slowing down the simulation.

Exporting a task from the target into the interface allows complicated transactions to complete in a single task call, with no switching between processes in the target and initiator (the target may not even contain any processes). An exported task can be called as if it were part of the interface (even though it is defined in a connected module).

Unfortunately, there is a problem if more than one target needs to export the same task to the interface—which task should be called? For example, consider the scenario in Figure 7:

```
interface Intf;
  int count;
  modport Bus  (output count,
                export CountMe);
  initial
    CountMe;
endinterface
```

```
module BusTarget (
          Interface bus);
  task bus.CountMe;
    bus.count++;
  endtask
endmodule
```

Which one gets called?

```
module Top;
  Intf TheBus();

  BusTarget BusTarget_inst1 (TheBus.Bus);
  BusTarget BusTarget_inst2 (TheBus.Bus);
endmodule
```

Two instances of BusTarget, each exports CountMe?

This example is erroneous

**Figure 7 - Problem of exporting the same task in an interface to multiple targets.**

In this example, it is unclear which target actually calls the exported task *CountMe*. The solution provided in SystemVerilog is to declare the task as an *export forkjoin* within the interface as shown in Figure 8.

```
interface Intf;
  int count;
  extern forkjoin task CountMe();
  modport Bus ( output count, export CountMe );
  initial
    repeat(10) CountMe;
endinterface

module BusTarget #( int id=0 ) ( interface bus );
  int seed = $random;
  task automatic bus.CountMe;
    int delay = $dist_uniform(seed,10*id,20*id);
    #(delay * 1ns);
    bus.count++;
    $display( "Count = %0d in %0d at %t",
              bus.count,id,$realtime );
  endtask
endmodule: BusTarget

module Top;
  Intf TheBus();
  BusTarget #(1) BusTarget_inst1( TheBus );
  BusTarget #(2) BusTarget_inst2( TheBus );
endmodule
```

**Figure 8 - Interface with extern forkjoin.**

When an *extern forkjoin* task in an interface is called, it is actually called once for each instance of the module defining it that is connected to the interface, just as if the task call was contained within a forked block.

Strictly speaking, *extern forkjoin* creates static rather than dynamic processes. However, we have started our discussion with them since there are some similarities with the way that dynamic processes are used in class-based testbenches.

### 2.1.2. Creating resolution functions in SystemVerilog

Along with dynamic processes comes dynamic process control, and process control has a particularly useful application when coupled with interfaces. Process control can be used to compensate for something plainly missing in SystemVerilog—the ability to define customized resolution functions for modeling analog wire behavior.

In a Verilog hierarchy, nets must be used to connect to the output ports of leaf components. One of the reasons for this requirement is that a net could be connected to (and therefore driven by) multiple output ports. When a simulation is run, the state of a net is calculated by "resolving" the contributions made by each of its drivers: for a wire the value will be either 0, 1, X or Z and each driver will be attempting to set that value with a driving strength of *supply*, *strong*, *pull* or *weak*. The effect of each combination of value and driving strength for multiple drivers is defined in the Verilog language reference manual. SystemVerilog interfaces also support connections to multiple output ports and offer us the ability to define our own resolution mechanism. To see how this might work, consider first how driver resolution works in VHDL.

VHDL takes a different approach from Verilog to resolving multiple drivers. VHDL *signals* are used exclusively for port connections, concurrent assignments and assignments within a process that need to be visible outside of the process (in other processes or as a port). No drive strength is associated with a signal assignment: instead the resolution is defined by a function associated with the type of value carried by a signal. Standard types such as *std_logic* have a built-in resolution function but VHDL also permits user-defined types with their own resolution functions to be used for signals. Such types open up possibilities to model signals in VHDL that do not have simple logic values or that have more complex rules to calculate the contribution made by each driver to the final value of a signal. A good example of this is to model a net whose resolved value is some function of the outputs from multiple drivers that generate real (floating point) values. This approach has been successfully applied to model mixed-signal effects in a VHDL simulator.

A VHDL package that contains a user-defined type (*a_type*) and its resolution function (*resolve*) is shown in Figure 9. In this case, the user-defined type is simply a real number that is associated with a function. Note that the argument to the resolution function is an unconstrained array of real values. The simulator creates this array automatically whenever it needs to calculate the value of a signal: each element of the array corresponds to the output produced by one of the drivers connected to the signal. Here, the resolution function is adding the values of each driver's contribution to calculate the result – a signal using this type will behave as a summing junction.

```
package a_pack is
  type real_vec is array (natural range <>) of real;
  function resolve (s: real_vec) return real;
  subtype a_type is resolve real;
end package;

package body a_pack is
  function resolve (s: real_vec) return real is
    variable sum: real := 0.0;
  begin
     for i in s'range loop
        sum := sum + s(i);
     end loop;
     return sum;
  end function resolve;
end package body;
```

**Figure 9 - VHDL package with resolved type.**

A VHDL driver for the resolved signal is shown in Figure 10. The driver is not aware that it is connected to a resolved signal (that is only checked at elaboration time). In this case, the driver is simply driving its output to one of two arbitrary real values but exactly the same approach could be taken to build a more useful component whose output was a complex waveform or some function of its inputs (e.g., a sine wave generator or filter, remembering of course that analog waveforms must be represented by a series of discrete steps in VHDL [3]).

```
use work.a_pack.all;

entity driver is
  generic (low, high : real := 0.0;
           delay : time := 1 ns);
  port (v: out a_type);
end entity driver;

architecture behav of driver is
begin
P1: process is
begin
  v <= low;
  wait for delay;
  v <= high;
  wait for delay;
  v <= low;
  wait;
 end process P1;
end architecture behav;
```

**Figure 10 - VHDL driver.**

Part of a VHDL testbench for the resolved signal is shown in Figure 11. Here, two drivers with different parameters have their outputs connected to a common signal (*a_wire*). The resolution function is called to calculate a new value for the signal whenever one of the driver outputs changes.

```
use std.textio.all;
use work.a_pack.all;

entity top is
end entity top;

architecture bench of top is

  component driver is
     generic ( low, high : real := 0.0;
               delay : time := 1 ns);
     port (v: out a_type);
  end component driver;

  signal a_wire : a_type;

begin

  D1: driver generic map (0.0, 2.5, 10 ns)
             port map (v => a_wire);

  D2: driver generic map (0.0, 1.25, 12 ns)
             port map (v => a_wire);
  ...
end architecture bench;
```

**Figure 11 - VHDL testbench for resolved signal.**

To emulate the behavior of the VHDL signal resolution mechanism described above requires the following steps:

1. Keep a record of the current contribution from every driver connected to the common signal.
2. Whenever a driver writes a new value, determine which driver is active (doing the write).
3. Update the contributions record entry for the active driver.
4. Calculate the result using the values in the contributions record.

These steps must be executed explicitly in SystemVerilog. However, if an interface is used as the common signal, a procedural modeling approach using an interface function can hide these steps from the model user. A suitable interface is shown in Figure 12.

```
interface analog_if ();
  real driver_vals[process];
  real result;

  function void resolve();
    //in this example just sums all values
    process index;
    result = 0.0;
    assert(driver_vals.first(index));
    forever begin
      result += driver_vals[index];
      if (driver_vals.next(index) == 0) break;
    end
  endfunction: resolve

  function void write (real val);
    process this_proc;
    this_proc = process::self;
    driver_vals[this_proc] = val;
    resolve();
  endfunction: write

endinterface: analog_if
```

**Figure 12 - SystemVerilog interface with "resolution" function.**

In this example, the driver contributions record is an associative array of *real* that uses a *process* class reference as its index. A driver calls the interface's write function to update its value. The write function calls *process::self* to get a reference to the currently active driver (step 2) – it then uses this reference to update the corresponding entry in the contributions record (step 3). Finally, the write function calls resolve to calculate the new result by summing the contribution of each driver in the record (step 4).

A simple driver module that works with the *analog_if* interface is shown in Figure 13, together with part of its testbench to show how the interface is connected. In this example, calls to *process::self* in the interface's write function will return a reference to the static initial procedure in each driver. However, many verification components are now built using SystemVerilog classes in preference to modules (using methodologies such as VMM, OVM and UVM). If the driver was a class rather than a module, the *process::self* would return a reference to a dynamic process rather than a static process but the operation of the interface would be identical in every other respect (although the interface would be accessed from within the class as a virtual interface rather than being passed as a port).

```
module driver #(real low=0.0, high=0.0,
                time delay=1ns)(analog_if aif);
  initial begin
    aif.write(low);
    #delay aif.write(high);
    #delay aif.write(low);
  end
endmodule: driver

module top;
  analog_if aif1();
  driver #(0.0,2.5,10ns)  D1(aif1);
  driver #(0.0,1.25,12ns) D2(aif1);
...
endmodule: top
```

**Figure 13 - SystemVerilog driver module and testbench.**

As Figure 13 demonstrates, interfaces can act as more than just a traditional wire, but as a creative way of modeling analog behavior—all made possible due to SystemVerilog's built-in dynamic process control.

## 2.2. Testbenches

Where dynamic processes find their greatest value and usefulness is in the realm of verification testbenches. A testbench is required to simultaneously provide all of the inputs to the design under test while monitoring all of its inputs and outputs, checking and logging the outputs, and recording coverage. In all but the most basic of testbenches, this requires multiple concurrent threads of execution (processes). In a Verilog module-based testbench, these threads would be static processes. In a SystemVerilog class-based testbench, these threads would be dynamic. We will consider a few practical applications for dynamic processes in testbenches in the following sections.

### 2.2.1. Testbench hierarchy

Class-based environments such as VMM, OVM, and UVM provide a set of base classes that are used as the starting point for reusable verification component classes and a testbench environment class. Among other things, these base classes define the mechanisms that are required in every component class to create a traversable hierarchy of components, to create parallel threads in those components, and to execute these threads at the correct points in a simulation.

A simplified component base class is shown in Figure 15. This class is not the same as in VMM/OVM/UVM, but is intended to show the main principles. In a component hierarchy (Figure 14), every component has exactly one parent and may have zero or more children. In our component class the links to parent and child components are held in member variables.

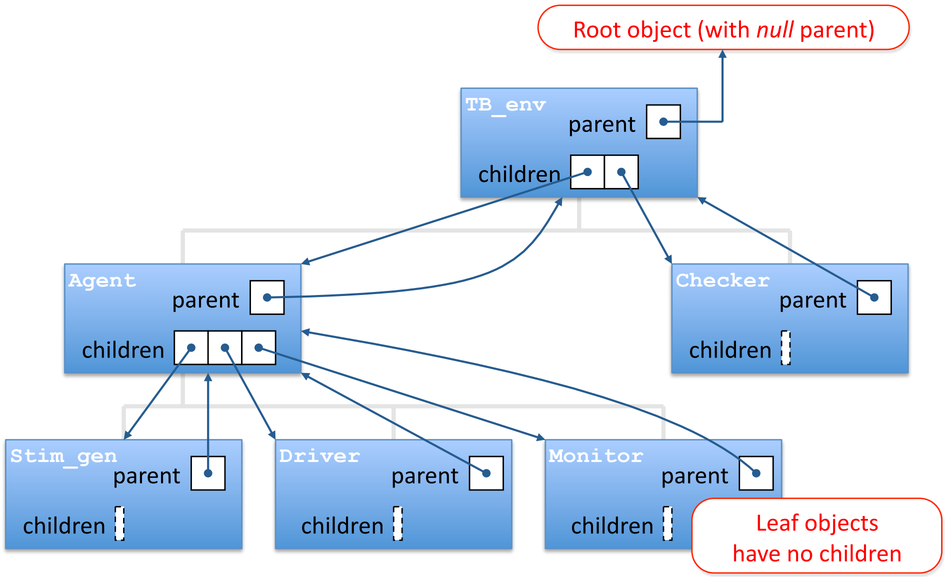


**Figure 14 – Class-based testbench hierarchy with parent-child relationship.**

```
virtual class Component;
  local     string    instance_name;
  local     Component parent;
  protected Component children[$];

  function new (string _instance_name, Component _parent);
    instance_name = _instance_name;
    parent = _parent;
    if (parent != null) parent.children.push_back(this);
  endfunction

  function Component get_parent();
    return parent;
  endfunction

  protected task m_do_run();
    foreach ( children[i] )
      children[i].m_do_run();

    fork
      run();            // run() becomes a dynamic process
    join_none
  endtask
  virtual task run();  endtask  //does nothing

endclass : Component
```

**Figure 15 - Simplified Component base class.**

A parent component is responsible for setting up a thread in each of its child components. In our component class, the thread corresponds to a virtual task named *run*. By default, our *run* task does nothing, but it may be overridden in any derived component class to implement the required behavior. The *run* task is turned into a thread by the component's *m_do_run* task – this is called recursively for every component in the hierarchy and spawns the thread by calling *run* within a *fork..join_none* statement.

The *m_do_run* task for the top-level component needs to be called at the start of simulation; otherwise, no dynamic processes will be created. A root component that extends our component class is shown in Figure 16. It contains a *run_test* task that calls *m_do_run* from inside another *fork..join none* and then waits until all spawned processes have completed (remember, *wait fork* waits for all children threads spawned as well).

```
class Root extends Component;
  ...
  task run_test();
   fork
     children[0].m_do_run();
   join_none;
   wait fork;
  endtask
endclass: Root
```

**Figure 16 – Root component class.**

An instance of the root class is at the top of the component hierarchy. It is instantiated in an *initial* procedure in the top-level module of the testbench. The *run_test* task is also called from within this *initial* procedure (there must be a static process to start up the dynamic processes). A very simple test module is shown in Figure 17.

```
module Test();
...
initial begin
  Root top = new( "top", null );
  Env env1 = new( "env1", top );       // Environment with
                                       // parent = top
  top.run_test();
  $finish;
end
endmodule
```

**Figure 17 - Starting the test from the initial procedure.**

Thankfully, when we use industry-standard class-based libraries like OVM/UVM/VMM, all these details have been implemented and abstracted away in base classes that automatically construct the hierarchy and kick of the component threads of execution in similar ways. Underlining these component classes is the basic SystemVerilog dynamic process, which makes it possible to construct such dynamic class-based testbench all at run-time.

### 2.2.2. Simulation phasing

Verification environments based on VMM, OVM or UVM use a more complex approach than the simple *run()* task presented above. They all split the simulation into a sequence of phases. Each phase has a corresponding "callback" function or task in the component base class that can optionally be overridden in an extended component class, if it needs to perform some actions during that phase. At the time of writing, UVM components include function-based phases such as *build*, *connect*, *end_of_elaboration*, *start_of_simulation* and *report* as well as a time-consuming *run* task (the function-based phases are not allowed to call *wait*).

For example, the following is a code snippet is taken from the *uvm_root* class illustrating the use of dynamic processes for simulation phasing (Figure 18):

```
// MAIN LOOP: Executes all phases from the current phase
// through the phase given in the argument
...
fork : task_based_phase
     m_stop_process();
     begin
        #0 m_do_phase(this,m_curr_phase);
        wait fork;
     end
     begin
        #timeout uvm_report_error("TIMOUT",
              $psprintf("Watchdog timeout of '%0t' expi-
red.", timeout), UVM_NONE);
     end
join_any
disable task_based_phase;
```

**Figure 18 - Simulation phasing in UVM.**

Notice how UVM uses a *fork..join_any* to spawn off the task-based phases, and even uses a *wait fork* within one of the forked processes. The *fork..join_any* provides a way to timeout if the forked process does not complete within the timeout limit. Whichever process finishes first results in disabling (killing) the other process. When the *m_do_phase()* method is spawned, an additional dynamic process is forked (if a task phase), which executes the component's virtual phase methods (Figure 19):

```
function void uvm_root::m_do_phase (uvm_component comp,
uvm_phase phase);
...
if (curr_phase.is_task()) begin
     // We fork here to ensure that do_task_phase, a
     // user-overridable task,
     // does not inadvertently block this process
     fork
        comp.do_task_phase(curr_phase);
     join_none
end
else
     comp.do_func_phase(curr_phase);
...
```

**Figure 19 - UVM phases fork additional processes.**

When phases are finished and simulation starts to shutdown, *disable fork* is used to end the component threads. Similarly, both OVM and UVM can use the *process* class for fine-grained process control on some simulators. While each class-library implements phasing in its own unique way, each phasing method is a great example of how dynamic processes can be applied to create simulation phasing.

### 2.2.3. Timeouts

Similar to simulation phasing, dynamic processes can be used for simulation timeouts. The *run_test()* task given in Figure 16 will return once all of the threads that it has spawned have completed. If there is any component in the hierarchy whose *run* task contains an infinite loop, then *run_test()* will never return – the simulation will carry on running until it is killed. It is often useful to be able to specify some upper limit on the simulation time to guard against such situations arising. A maximum timeout can easily be added to the *Root* class, as shown in Figure 20.

```
class Root extends Component;
  int unsigned timeout = -1;  //a big number!
  ...
  task run_test();
   fork
     begin
       fork
         children[0].m_do_run();
       join_none;
       wait fork;
     end
     begin
       #timeout;
       $display("Max simultion timeout reached!");
     end
   join_any
   disable fork;
  endtask
endclass: Root
```

**Figure 20 - run_test task with timeout.**

The timeout is waited for in a dynamic process that runs in parallel with the main *run* task: the dynamic process is created by a *fork..join_any* so the main thread continues as soon as one of the branches (the set of all *run* tasks or the timeout) have completed. A *disable fork* statement immediately follows the *fork join_any*. This kills the remaining dynamic processes (e.g., stopping all *run* threads once the timeout has been reached). The timeout can be set in the top-level module as shown in Figure 21.

```
module Test();
...
initial begin
  Root top = new( "top", null );
  Env env1 = new( "env1", top );       // Environment with
                                       // parent = top
  top.timeout = 50;   // Set timeout before running test
  top.run_test();
  $finish;
end
endmodule
```

**Figure 21 - Setting the test timeout.**

OVM and UVM use a similar (although slightly more complex mechanism). They also include other ways of stopping the simulation such as an objection mechanism.

### 2.2.4. Communicating outside of SystemVerilog using adapters

Another useful application for dynamic processes is communicating with the world outside of SystemVerilog. Many languages and tools go into developing and verifying any design. High-level models are built for architectural modeling, virtual platforms, performance analysis, or verification. Stimulus may be created from automated tools, DSP models (like MATLAB), SystemC/C++/C programs, or even sophisticated Tcl/Perl/Python scripts. Fortunately, System/Verilog has always provided an interface for linking in other programs through originally PLI/VPI and now DPI (the Direct Programming Interface).

In order to communicate with the foreign code through the interface, a SystemVerilog process must either call the code or wait for a response. This would tie up a process as it communicates across the foreign interface. Instead, a dynamic process can be used to service the communication across the foreign interface and coordinate with the rest of the testbench. If the communication needs translated from one form to another across the interface, then the dynamic process can be thought of as an *adapter* (Figure 22).

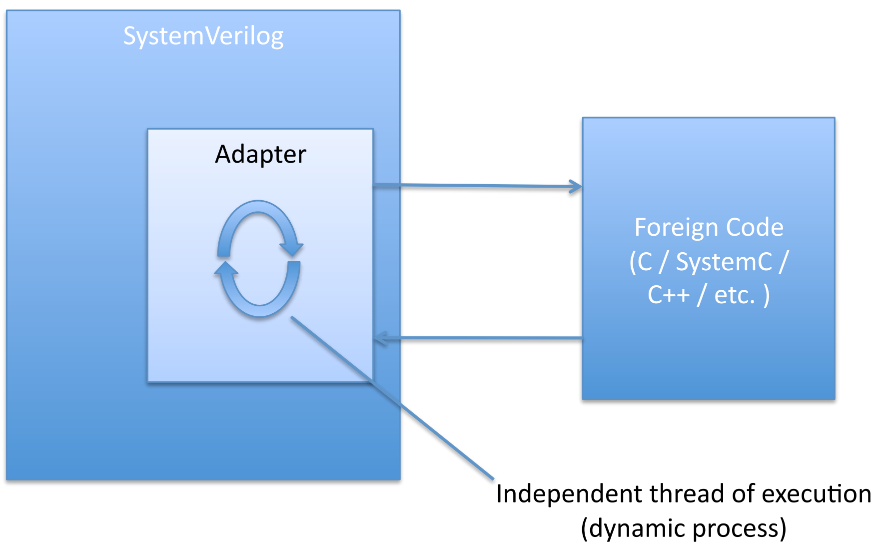


**Figure 22 - Dynamic processes used as an adapter interface with foreign code.**

A good example of such an adapter is Synopsys' Transaction Level Interface (TLI) [5]. The TLI is used to communicate transactionally between a SystemVerilog testbench (e.g., VMM) using TLM 2.0 connections to a high-level SystemC TLM2 model. Between the SystemVerilog testbench and SystemC model sits an adapter that processes communication back-and-forth between the two domains. On the SystemVerilog side, a *fork..join_none* is used to spawn a dynamic process when the TLM port is bound, which puts or gets the requests and responses across the connection. TLI also uses a *fork..join_none* to spawn off various DPI functions used to communicate between a VMM channel and SystemC. For example, a sample code snippet from the TLI-3 *tli_sv_bindings.sv* file is given in Figure 23.

```
function tli_tlm_bind(vmm_tlm_base tlm_intf,
vmm_tlm::intf_e port_type , string port_name);

  ...
  case(port_type)

     vmm_tlm::TLM_NONBLOCKING_EXPORT:
     begin
        ...
        // Since this is a blocking export, a process is
        // forked off to call the function b_transport()
        fork
           call_transport_process();
        join_none

     end
```

**Figure 23 – TLI binding function.**

A very practical use of such an adaptor would be to integrate external stimulus from a standalone program and bring it directly into a SystemVerilog testbench. For example, legacy programs or scripts written in Tcl, Perl, or Python are often used to generate vectors, commands, or even assembly code as in the case of a microprocessor. Programs such as these are tried and tested often with many years of development effort, and require significant re-work to re-implement inside a SystemVerilog environment. In such cases, the more prudent choice may be to simply pull in the output of such programs through the DPI interface and pass on the stimulus into a SystemVerilog testbench (Figure 24).

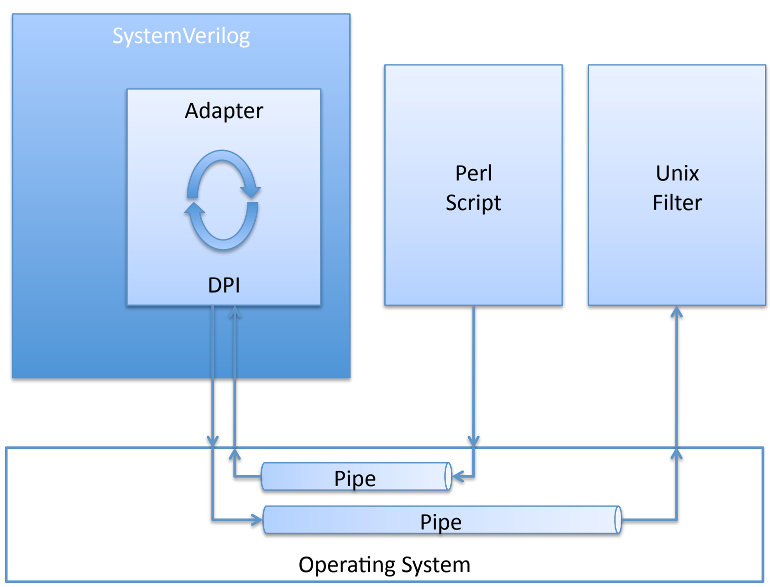


**Figure 24 - Dynamic processes used for interprocess communication with external programs.**

Using dynamic processes to call DPI functions, an external program can communicate with the DPI through interprocess communication (IPC) such as an operating system pipe. Inside the DPI code, the POSIX *popen()* method spawns a shell, executes a command, and pipes the standard output back into SystemVerilog. For example, suppose a Perl script is used to randomly generate an instruction stream to verify a microprocessor or exercise a system-on-chip. To illustrate, a simple Perl script might look like Figure 25 (ignoring instruction operands for simplicity sake):

```
#!perl
# Filename:  perlinput.pl

@operate_ops = ( "add", "sub", "or", "and", "xor" );
@branch_ops  = ( "jmp", "be", "ble", "bge", "bg" );
@store_ops   = ( "ld", "st" );
@all_ops     = ( @operate_ops, @branch_ops, @store_ops );

for ( $i = 0; $i < 100; $i++ ) {
   my $index = rand @all_ops;
   print $all_ops[$index], "\n";  #Pick random instructions
}
```

**Figure 25 - Perl script called from SystemVerilog.**

The DPI code invokes the Perl script using *popen()* and reads in the text input from the script's standard output. The *popen()* function actually creates a dynamic process in C, spawning a simple shell, executing the command, and passing the input or output. For two-way communication to send a responses back to the external program, a streaming pipe, message queue, or socket could be used, but a single pipe is used in this example to keep it simple (see [4]for details on full-duplex interprocess communication). The DPI will also spawn a process to filter the output from the design under test and redirect the filtered output into a file. The code is given in Figure 26.

```
#include <stdio.h>
#include <stdlib.h>
#define PERLCMD  "./perlinput.pl"         // Perl script
#define FILTERCMD  "grep JMP > jmp.txt"   // Filter command
#define MAXLINE 256
FILE *fpin, *fpout;
char line[MAXLINE];

int open_pipe() {
   // Open the pipe with the Perl script
   if ( (fpin = popen( PERLCMD, "r" )) == NULL ) {
        fputs( "Cannot open perl script pipe!", stderr );
        return 1;
   } else {
        // Discard initial line of garbage
        fgets( line, MAXLINE, fpin );
        return 0;
   }
}
int open_filter() {
    if ( (fpout = popen( FILTERCMD, "w" )) == NULL ) {
         fputs( "Cannot open filter pipe!", stderr );
         return 1;
    }
    return 0;
}
char *get_stimulus() {
    // Open pipe if not yet created
    if ( fpin == NULL )
       if ( open_pipe() ) return NULL;

    // Read in the stimulus
    if ( fgets( line, MAXLINE, fpin ) != NULL )
       return line;        // Send to the DPI

    return NULL;
}
void filter( char *l ) {
    if ( fpout == NULL )
       if ( open_filter() ) return NULL;

    if ( !fputs( l, fpout ))
       fputs( "fputs error!", stderr );
}
```

**Figure 26 - C code to call Perl script.**

Notice, the function *get_stimulus()* returns one line of text at a time from the external Perl script. The *filter()* function spawns the filter command (i.e., the Unix command of `'sh -c "| grep JMP > jmp.txt"'`) and sends the string it receives to the command. This command finds any jump instructions and redirects them to an output file called *jmp.txt*. Since an external program runs independently of the SystemVerilog testbench, a forked process is used to read in the input until it finishes while performing the filtering in a separate dynamic process. The code is given in Figure 27.

```
import "DPI-C" function string get_stimulus();
import "DPI-C" function void filter(input string line);

class adapter;
    ...
    task run();

        fork
            // Process 1 - Read in stimulus from Perl script
            begin
                string  op, in;
                while ( ( in = get_stimulus()) != "" ) begin
                        // Remove newline
                        assert( $sscanf( in, "%s", op ));

                        // Create instruction to send the DUT
                        instr_t  tr = new ( op );

                        // Send instruction transaction ...
                        ...
                    end
            end

            // Process 2 - Filter output from DUT through
            // external command
            begin
                forever begin
                        // Get transaction and send to filter
                        ...
                        filter( tr.sprint() );
                        ...
                    end
            end
        join_any
```

```
        // Do other work here.  Decide if it is time to end.
        ...

        // End the all processes
        disable fork;
    endtask
endclass
```

**Figure 27 - SystemVerilog adaptor using DPI.**

The forked processes enable the use of *forever* to continuously process output sent to the Unix filter. Once the Perl input finishes, the *run()* task picks up after the *fork..join_any* statement and either does more work or decides if the design is finished sending output and thus, kills the forked process using *disable fork*. With such an approach, dynamic processes make creating adapters easy to implement in SystemVerilog, which greatly simplifies interfacing with foreign code and external programs.

## 2.3. Stimulus

One of the most common dynamic process applications is the creation of random stimulus. With random stimulus, we can quickly explore our design's state space, and with dynamic processes we can generate parallel threads of execution that bombard a design with input in order to reach interesting corner cases. In the following sections, two practical applications of dynamic processes will be considered.

### 2.3.1. Sequences

OVM and UVM sequences are executed as dynamic processes by their sequencer components. The use of dynamic processes allows a sequence to be halted before it has reached its final step (by calling *disable fork*). It is also sometimes useful to spawn new dynamic threads within a sequence. A typical example is where a "virtual sequence" (a sequence that controls several other sequences) needs to start sequences on several different sequencers simultaneously, without waiting for each one to complete. An example of doing just that in UVM is shown in Figure 28. Here, the two initialization sequences are run simultaneously on two separate sequencers that are being controlled by the virtual sequence. The virtual sequence then waits (*wait fork*) until both initialization sequences are complete before running a sequence on *sequencer0* followed by a sequence on *sequencer1*.

```
class my_virtual_sequence extends uvm_sequence;

`uvm_sequence_utils(my_virtual sequence,my_v_sequencer)

init_sequence0 iseq_inst0;
init_sequence1 iseq_inst1;
exec_sequence eseq_inst0;
exec_sequence eseq_inst1;

...
virtual task body();
  fork
    p_sequencer.sequencer0.start(iseq_inst0);
  join_none
  #0;  //wait allows thread to start

  fork
    p_sequencer.sequencer1.start(iseq_inst1);
  join_none
  wait fork;
  `uvm_do_on(eseq_inst0, p_sequencer.sequencer0)
  `uvm_do_on(esqe_inst1, p_sequencer.sequencer1)
endtask: body

endclass: my_virtual_sequence
```

**Figure 28 - UVM Virtual Sequence with Dynamic Threads.**

Notice the use of *#0* between the *fork..join_none* statements. The small delay ensures that the dynamic process to run the sequence on *sequencer0* starts before execution in the virtual sequence body continues. Once *sequencer0* starts, the next sequence is started on *sequencer1*. In order to synchronize with both dynamic threads, the *wait fork* is used. In this way, virtual sequences can create and control the execution of sequences within a testbench environment.

### 2.3.2. Random stability

Each process that makes a call to *std::randomize* or the randomize function of a class contains its own local random number generator (RNG). The sequence of random numbers returned within one thread is independent of that returned in any other thread. When a dynamic process is created, its RNG is seeded with the next random number from its parent process. A small change to the code can sometimes result in dynamic threads being created in a different order and therefore getting different seeds. This can produce big differences in the stimulus values generated. To preserve the set of random values generated, the order that threads are created in each branch of the thread tree must be maintained.

Each object also has its own RNG. When an object is created, its RNG is seeded from the next random number in the thread that calls *new*. When generating a sequence of randomized objects (e.g. transactions), better random stability is obtained by creating copies of a single "template" object that is repeatedly randomized, rather than multiple objects that are each only randomized once. Examples of object generators with good and bad random stability are given in Figure 29.

```
class Stream_Gen_Bad;
  function Trans get_trans();
    Trans t;
    t = new(this);         // BAD - new during run-time
    void'(t.randomize());
    return t;
  endfunction: get_trans

endclass: Stream_Gen_Bad

class Stream_Gen_Good;
  Trans template;
  function new();
    template = new(this);    // GOOD - new at beginning
  endfunction

  function Trans get_trans();
    Trans t;
    void'(template.randomize());
    t = template.copy();
    return t;
  endfunction: get_trans

endclass: Stream_Gen_Good
```

**Figure 29 - Stream generators with good and bad random stability.**

When exploiting the many uses of dynamic processes, it is important to follow this template approach above to guarantee random test stimulus reproducibility.

## 2.4. Architectural and Behavioral Modeling

Creating independent threads of execution that can communicate with each other is crucial for architecturally modeling designs and systems-on-chip. Using dynamic processes to model analog behavior has been demonstrated in section 2.1.2, and using dynamic processes to construct components like those used in architectural models has been shown in the testbench section 2.2.1. Another issue that arises when creating architectural models is accessing shared memory between independent threads of execution. Again, dynamic process control is ideal for eliminating these issues, which we will examine in the following section.

### 2.4.1. Shared memory mutex

The use of shared memory between parallel processes may result in a common multi-threaded programming issue. For example, a common shared memory situation might arise inside of a testbench scoreboard. As expected values are sent to the scoreboard, they are inserted into a queue and removed from the queue when the actual values arrive. Given the way SystemVerilog evaluates processes, as long as neither process reads or writes in a time-consuming way, then shared memory issues are generally avoided. However, if either process waits or consumes time then there is the potential for values to become invalid or lost. Using a shared memory mutex around the critical read and write regions would solve any potential issues.

The SystemVerilog standard defines a built-in semaphore class to provide mutual exclusion in cases like sharing a common memory location. However, the semaphore class has no built-in checking to ensure that it is used properly. For example, it is possible to put more keys into the semaphore than when it was created, and it is possible to return keys from processes even if the process did not get the key.

Given that, wrapping a semaphore object in another class is a good idea in order to enforce a safe usage policy, and using the built-in process control from the *std::process* class is a great way to accomplish it. Since *std::process* can return a handle to the current process by calling *self()*, this handle can be used to verify that the same process returns the key (lock) that was granted the key. The code in Figure 30 shows how the process' handle used in an associative array keeps track of the number of keys each process has been granted.

```
class mutex;
    semaphore    lock;
    int          granted[ process ];

    function new( int keys = 1 );
       lock = new( keys );
    endfunction : new

    task put( int keyCount = 1 );
       process index = process::self;

       assert( granted[index] >= keyCount ) else
       begin
          $error( "Process not granted the key(s)!");
          return;
       end
       granted[index] -= keyCount;
       lock.put( keyCount );
    endtask : put
```

```
    task get( int keyCount = 1 );
       process index = process::self;
       lock.get( keyCount );
       granted[index] += keyCount;
    endtask : get

    function int try_get( int keyCount = 1 );
       process index = process::self;

       if ( lock.try_get( keyCount ) ) begin
          granted[index] += keyCount;
          return keyCount;
       end
       else return 0;           // Failure to get a key
    endfunction : try_get
endclass : mutex
```

**Figure 30 - Mutex class.**

The mutex class simply re-implements the semaphore class' methods with some additional checking of the process ID. Additional policies also could be implemented like having separate read and write keys, allowing multiple read keys while restricting write keys, etc. As with modeling resolution functions, the dynamic process control of SystemVerilog provides the necessary framework to solve the many types of modeling problems engineers face.

# 3 Conclusions

The creation of dynamic processes and fine-grain process control are key features in building testbenches capable of meeting the verification and modeling challenges commonly faced today. They provide run-time creation and destruction of parallel threads of execution as well as process identification—a feature with several practical modeling applications. While Verilog's traditional static processes are useful, SystemVerilog's enhanced dynamic processes raise what can be done in a Verilog testbench to a much more sophisticated level. This paper highlights only a sampling of the many practical applications for dynamic processes. When face with your next verification or modeling challenge, consider if dynamic processes might provide you with that next ideal solution.

# 4 References

[1] IEEE Std 1800™-2005. IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language. IEEE Computer Society, New York, 2005.

[2] IEEE Std 1800™-2009. IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language. IEEE Computer Society, New York, 2009.

[3] Long,D. I. "Behavioural Modelling of Mixed-Signal Circuits using PWL Waveforms", IEE Colloquium on Mixed Signal AHDL/VHDL Modelling and Synthesis, London, 19 November, 1997.

[4] Stevens, W. Richard. Advanced Programming in the UNIX Environment. Addison-Wesley, (Reading, Massachussets) 1993.

[5] VCS TLI Adapter User Guide. Synopsys, November 2009.